Scientific Time Sharing Corporation
7316 Wisconsin Avenue
Bethesda, Maryland 20014
(301) 657-8220

Working Memorandum No. 118

WRITING INTERACTIVE APL PROGRAMS:

A DISCUSSION OF TECHNIQUES

January 1976
Revised: August 1976

by
Martin A. Zimelis

Special thanks to Howie Bryks, Jim Davis, Brian Hagenbuch, Eric Landau, Bernard Stowens, Rex Swain, and Roy Sykes, all of Scientific Time Sharing Corporation, whose contributions and helpful criticism made the writing of this memorandum possible.

This document was prepared to assist users of Scientific Time Sharing Corporation's *APL*PLUS* Time Sharing Service; its contents may not be used for any other purpose without written permission. The material contained herein is supplied without representation or warranty of any kind. Scientific Time Sharing Corporation therefore assumes no responsibility and shall have no liability of any kind arising from the supply or use of this document or the material contained herein.

Copyright © 1976 Scientific Time Sharing Corporation

Printed in U.S.A.

All rights reserved, including the right to reproduce this document or any portion thereof in any form.

*APL*PLUS* is a registered service mark of Scientific Time Sharing Corporation.

Selectric is a registered trademark of IBM Corporation.

WRITING INTERACTIVE APL PROGRAMS:

A DISCUSSION OF TECHNIQUES

TABLE OF CONTENTS

Introduction	1
What Is an Interactive Program?	2
Developing a Basic Prompting Function	3
Prompting for Numeric Input	12
Prompting for a Specific Response	16
Data Security	18
Restarting Functions	21
Validating Input	23

Introduction

This memorandum gives an overview of the most commonly used types of interactive APL programs. Emphasis is placed on the design and implementation of modular subroutines so that the user may quickly develop his own suite of simple functions for prompting, input validation, and error reporting which may be transported from one application to another with minimal need for modification.

A typical prompting program is developed from an extremely simple form to a fairly complex routine which allows multiple levels of prompts, performs initial error checking, and allows for certain terminal hardware idiosyncrasies. The discussion addresses functions that prompt for character input, those that prompt for numeric input, and those that specify a particular set of valid responses.

Techniques for incorporating simple debugging aids into the system, restarting a program or program package after a line drop, and building security measures into a system rather than adding them on afterward are also discussed.

It should be noted that this memorandum was originally written with the intent that it be supplemented with an oral presentation in a seminar environment. All of the sample functions used in this memorandum can be found in workspace 902 *INTERACTIVE* on the *APL*PLUS* System.

What Is an Interactive Program?

The primary question is "What is an interactive program?". There is no formal definition of the term "interactive program" and it covers such a wide variety of programs that it may be best to define it with an example.

Basically, interactive programs are of two major varieties: data entry programs, in which the terminal acts as an intelligent keypunch; and what might be called, for lack of a better name, information retrieval programs. The latter are the more interesting and are the form dealt with most extensively in this memorandum.

Examples of data entry applications include general ledger journal entry with pre-editing for format and reasonableness of data; for example, the database entry phase of the *PERT*PLUS* System, during which the user enters all the links in the critical path database. In this case, there is little or no prompting from the system if the line typed by the user is correct as far as its syntax is concerned. Checks on reasonableness of data are sometimes done at this time but this varies from application to application. In the *PERT*PLUS* System, substantial error checking is done with the input of each line.

Information retrieval applications have three major subsections: inquiry, in which information is drawn from a database without changing its contents; update, in which some of the contents of the database are changed according to user inputs; and report generation, which is often a result or final output of the inquiry process. Certain interactive program packages, such as the *APL*PLUS* Message Processing System (Mailbox), combine several of the subsections in a single action. Similarly, a widely-used function, *TEACH* in workspace 1 *APLCOURSE*, exhibits many of the properties of interactive programs. With each interaction, data (the user's score on each primitive function) are collected. Simple error checking is performed and a diagnostic message printed in case of error (*TRY AGAIN*). A report is generated as a result of the final user input (*STOP*).

Developing a Basic Prompting Function

Prior to the discussion of prompting functions, it is necessary to briefly discuss how quote-quad input and output (I/O) work on the *APL*PLUS* System. Currently, each workspace has a 250-character buffer which is used for terminal output. When a user specifies quote-quad output, the desired characters are placed in the buffer. They remain in the buffer until one of several things happens: (1) raw (non-quote-quad) output to the terminal is performed, (2) the buffer fills up, (3) input is requested, or (4) another APL function causes the buffer to be flushed. Functions that cause the buffer to be flushed include *ΔWS*, *ΔFD*, *ΔFMT*, and *ARBOUT*, all available in workspace 1 *WSFNS*, and system functions such as *⎕ARBOUT* and *⎕FMT*.

What is desired is to print a prompt and accept a response on the same printed line on the terminal. Quote-quad I/O, by itself, has the following side effect:

```
      ⍞←'PROMPT ' ⋄ R←⍞
PROMPT RESPONSE
      R
PROMPT RESPONSE
```

The entire line is accepted as the response. The desired effect is to dispose of the prompt and keep only the true response as input. One way to do this is with an invisible delimiter character at the end of the prompt. For instance, *⎕TCNUL*, the "idle" character, is a good candidate for this honor. A function incorporating this technique would look something like this:

```
⍞←'PROMPT ',⎕TCNUL ⋄ R←⍞ ⋄ R←(R⍳⎕TCNUL)↓R
```

This technique has several problems. Because the buffer contains the prompt, the system recognizes and accepts backspaces up to the actual left margin (i.e., over the prompt). It is then possible for the user to accidentally or, in a perverse mood, intentionally backspace over the prompt, press the LINEFEED (or ATTN) key, and erase part of the prompt -- including the delimiter character. Thus, if this algorithm is used, the final result in *R* is an empty character vector. Another problem also becomes evident if this method is used. Although the idle character does not print, it <u>is</u> included in the character count used by the system in determining cursor position. Thus, the cursor's theoretical and actual positions are out of sync. This presents no problems unless the user insists on using (or mistakenly uses) the TAB key in his response.

Probably the easiest way to circumvent all these problems is to use one of the APL functions that flushes the workspace output buffer. The best candidate for this is *⎕ARBOUT* because it is the only one of the functions named that does not force any carriage returns. Since *⎕ARBOUT* shouldn't print anything of its own, one should specify *⎕ARBOUT* ⍳0 to flush the buffer. Hence,

```
      ⍞←'PROMPT ' ⋄ ⎕ARBOUT ⍳0 ⋄ R←⍞
PROMPT RESPONSE
      R
RESPONSE
```

gives rise to the simplest of prompting functions:

```
    ∇ Z←PR1 P
[1]   ⍞←P ⋄ A0 ⋄ Z←,⍞
    ∇
```

Note that in this and all successive examples, the function *A0* is merely a niladic function that executes ⎕*ARBOUT* ⍳0. The functions *IF*, *THISL*, and *NEXTL* are defined in the usual manner (see workspace 902 *INTERACTIVE*).

There are several ways to embellish this simplest of prompting functions:

```
    ∇ Z←PR1A P
[1]   ⍞←P ⋄ A0 ⋄ Z←' ' SQ⍞
    ∇
```

The function *SQ* "blank-squeezes" the response (see below), thus ensuring that the response starts in the first character of the return variable.

```
    ∇ Z←C SQ X
[1]   X←X,C ⋄ Z←X≠C ⋄ Z←Z∨0,¯1↓Z ⋄ Z←¯1↓Z/X
    ∇
```

One of the side effects of flushing the workspace buffer with *ARBOUT* is that the system thinks the cursor is at the left margin. Hence, character counts and tabbing work on that assumption. If the keyboard does not unlock exactly on a tab stop, use of the TAB key causes undesirable results. It can be ensured that the response starts on a tab stop by modifying the function so that it issues a "tab" character via ⎕*ARBOUT* instead of nothing.

```
    ∇ Z←PR1B P
[1]   Z←×1↑3↓2 ΔWS 3 ⋄ ⍞←P ⋄ ⎕ARBOUT(Z× 0 1=2≥⌶28)/9 61 ⋄
      Z←,⍞
    ∇
```

Any of these functions works very nicely for simple interactive applications. However, it is easy to build in some additional features that greatly enhance its usefulness. For instance, it's annoying to have to remember to put a space (or colon, or question mark) at the end of each prompt. So, let the function do it:

```
    ∇ Z←PR2 P
[1]   P←P,(' '≠¯1↑P)/' ' ⍝ ENSURE A SPACE BETWEEN PROMPT
      AND RESPONSE
[2]   ⎕←P ⋄ A0 ⋄ Z←,⎕
    ∇
```

To all appearances, this is a perfectly adequate prompting function. It handles output and input on the same line; it produces easily readable hard copy; it provides a bit of security because it only uses quote-quad input -- only quote-quad! Anyone who has ever tried, for debugging purposes, to interrupt a function that accepts a lot of quote-quad input knows it's difficult, at best. So, the next step is to build in a facility by which the function may be suspended. The use of this facility is best explained with an example.

```
    ∇ Z←PR3 P
[1]   P←P,(' '≠¯1↑P)/' ' ⍝ ENSURE A SPACE BETWEEN PROMPT
      AND RESPONSE
[2]   ⎕←P ⋄ A0 ⋄ Z←,⎕
[3]   →0 IF '🁣'≠1↑Z ⋄ ⎕←'INTENTIONAL ' ⋄ A0 ⍝ DEBUGGING AID
[4]   RESUME VIA  →1
    ∇
```

Since it is very difficult to interrupt a function that prompts for several successive lines of input, a programmed error stop is built in. If the first character of the user's response is a domino (🁣), the function suspends after issuing the message *INTENTIONAL SYNTAX ERROR*.

```
      PR3 'ENTER NAME AND ADDRESS'
ENTER NAME AND ADDRESS 🁣
INTENTIONAL SYNTAX ERROR
PR3[4] RESUME VIA→1
                ^
```

Now the programmer is free to examine local variables at an intermediate point to see if the function is doing what is intended. An alternative method for suspending a prompting function is to not assign (or erase via ⎕*ERASE*) the return variable, producing a *VALUE ERROR* in the calling function. The only problem with this is that it assumes the calling function is restartable on that line. This is not always true. The method used in *PR3* also has a problem: the identifiers used in the prompting routine may coincide with some of those used in the calling function. Thus, the more globally-defined values are said to be "shadowed" and cannot be displayed. The solution to this is to use bizarre-named identifiers in the subroutine. (An often-used convention is to start at Δ99 and number backward for as many symbols as are needed.)

Problems aside, this function has a stylistic flaw: it requires a branch to a line number other than 0 (rather than a branch to a label). Besides, there is no need to check the prompt for trailing spaces again. Therefore, all that is

necessary to make it more esthetically pleasing is to put a label of, say, *GOON* on line 2 and change the comment on line 4.

```
      ∇ Z←PR3A P
[1]     P←P,(' '≠¯1↑P)/' ' ⍝ ENSURE A SPACE BETWEEN PROMPT
        AND RESPONSE
[2]   GOON:⍞←P ⋄ A0 ⋄ Z←,⍞
[3]     →0 IF '⍎'≠1↑Z ⋄ ⍞←'INTENTIONAL ' ⋄ A0 ⍝ DEBUGGING AID
[4]     RESUME VIA  →GOON
      ∇
```

This function is a little neater and does slightly less work than the previous one.

From this point on, all changes made to the function are primarily decorative rather than altering what it does.

Another addition that can make a programmer's life easier is a means to bypass the quote-quad. When testing an application, it is fairly common practice to use the same test cases over and over. Some of the responses to prompts may be lengthy or difficult to type. To eliminate the need to retype the responses each time, one might build a feature like the following into a prompting function:

```
      ∇ Z←PR4 P
[1]     P←P,(' '≠¯1↑P)/' ' ⍝ ENSURE A SPACE BETWEEN PROMPT
        AND RESPONSE
[2]   GOON:⍞←P ⋄ A0 ⋄ Z←,⍞
[3]     →NEXTL IF '⎕'≠1↑Z ⋄ Z←⎕ ⋄ →0 ⍝ ALLOW MASS INPUT
[4]     →0 IF '⍎'≠1↑Z ⋄ ⍞←'INTENTIONAL ' ⋄ A0 ⍝ DEBUGGING AID
[5]     RESUME VIA  →GOON
      ∇
```

The function checks for "⎕" as the first character of the input line. If there is a match, evaluated input is opened. This allows the person debugging the program to store standard responses in global variables and to simply type the name of the variable instead of the response itself. For example, the following sequence of statements might be entered:

```
        RESP1←'MARTIN ZIMELIS⍎7316 WISCONSIN AVE⍎BETHESDA, MD'
        ⍝ FIRST TEST CASE
        PR4 'ENTER NAME AND ADDRESS'
ENTER NAME AND ADDRESS ⎕
⎕:
        RESP1
```

For the sake of consistency (that is, a character-valued result is always expected from the prompting function), the format primitive ⍕ should be added to the left of the quad in the latest version of the function:

```
    ∇ Z←PR4A P
[1]   P←P,(' '≠¯1↑P)/' ' ⍝ ENSURE A SPACE BETWEEN PROMPT
      AND RESPONSE
[2]  GOON:⎕←P ⋄ A0 ⋄ Z←,⍞
[3]   →NEXTL IF '⎕'≠1↑Z ⋄ Z←⍎⎕ ⋄ →0 ⍝ ALLOW MASS INPUT
[4]   →0 IF '⍞'≠1↑Z ⋄ ⍞←'INTENTIONAL ' ⋄ A0 ⍝ DEBUGGING AID
[5]   RESUME VIA  →GOON
    ∇
```

Up to this point, one side effect of quote-quad/*ARBOUT* I/O has been completely ignored. This is only important on those terminals that are unbuffered <u>and</u> have a relatively slow (less than 25 cm/sec) carrier return rate. The effect is printing "on the fly" while returning to the left margin. One solution is to "idle" or delay a sufficient amount of time for the slowest terminal used on the *APL*PLUS* System to return to the left margin. Since the return rate is constant for any given terminal, "return time" is a function of the distance from the left margin. The side effect is introduced because of the use of *ARBOUT* to flush the workspace buffer. The system thinks the carrier is at the left margin when, in fact, it is *ρPROMPT* characters distant from it. The amount of time to be delayed then is simply ⌊.1×(*ρPROMPT*)+*ρRESPONSE* . The function *PR5* does just that:

```
    ∇ Z←PR5 P;R
[1]   R←ρP←P,(' '≠¯1↑P)/' ' ⍝ ENSURE A SPACE BETWEEN PROMPT
      AND RESPONSE
[2]  GOON:⎕←P ⋄ A0 ⋄ R←⌊(R+ρZ←,⍞)÷10 ⋄ ⎕ARBOUT Rρ47×2≥⍳28
      ⍝ ALLOW SUFFICIENT IDLES FOR RETURN
[3]   →NEXTL IF '⎕'≠1↑Z ⋄ Z←⍎⎕ ⋄ →0 ⍝ ALLOW MASS INPUT
[4]   →0 IF '⍞'≠1↑Z ⋄ ⍞←'INTENTIONAL ' ⋄ A0 ⍝ DEBUGGING AID
[5]   RESUME VIA  →GOON
    ∇
```

(Note that 47 is the *ARBOUT* code for the idle character on Selectric-based terminals and 0 is the code on ASCII terminals.)

One point generally recognized is that users come with differing levels of expertise in responding to a terminal. Thus, it may be desirable to allow for multiple levels of prompts in a given interaction; that is, each level of prompting is more verbose than the previous one. The function *PR6* incorporates just such a scheme:

```
    ∇ Z←PR6 P;A;B;C;R
[1]  GOON:A←P
[2]  L1:C←⍴B←((A⍷'⍝')-⍳1)↑A ⍝ SPLIT OUT FIRST PROMPT
       IN STRING
[3]    B←B,(' '≠¯1↑B)/' ' ⍝ ENSURE A SPACE BETWEEN PROMPT
       AND RESPONSE
[4]    ⍞←B ⋄ A0 ⋄ R←⌊(C+⍴Z←,⍞)÷10 ⋄ ⎕ARBOUT R⍴47×2≥⍳28
       ⍝ ALLOW SUFFICIENT IDLES FOR RETURN
[5]    →NEXTL IF 0≠⍴Z ⋄ A←((C≠⍴A)×1+C)↓A ⋄ →L1 ⍝ POP A
       LEVEL OF PROMPT (UNLESS ITS THE LAST ONE)
[6]    →NEXTL IF '⎕'≠1↑Z ⋄ Z←⍎⎕ ⋄ →0 ⍝ ALLOW MASS INPUT
[7]    →0 IF '⍞'≠1↑Z ⋄ ⍞←'INTENTIONAL ' ⋄ A0 ⍝ DEBUGGING AID
[8]    RESUME VIA  →GOON
    ∇
```

In this example, all levels of the prompt are passed to the function as a single character vector with individual levels separated by lamp (⍝) symbols. Note that the next most verbose level of prompt is elicited by responding to the prompt with only a RETURN.

```
      PMT1←'N/A⍝NAME AND ADDR⍝ENTER YOUR NAME AND ADDRESS'
      PR6 PMT1
N/A [RETURN]
NAME AND ADDR [RETURN]
ENTER YOUR NAME AND ADDRESS ⎕
⎕:
      RESP1
```

A few nice spin-offs of this scheme are fairly apparent. First, a full-blown multilevel prompting system is going to cost heavily in workspace used. Solution: put the prompts on file and pass the number of the prompt (which may or may not correspond to the component number) to the function rather than the prompt itself. The next example accepts either a prompt number or the prompt text, depending on the data type of the argument.

```
    ∇ Z←PR6A P;A;B;C;R
[1]    →GOON IF ' '=1↑0⍴P ⋄ P←PREAD P ⍝ PROMPT IS FILED.
       RETRIEVE IT
[2]  GOON:A←,P
[3]  L1:C←⍴B←((A⍷'⍝')-⍳1)↑A ⍝ SPLIT OUT FIRST PROMPT
       IN STRING
[4]    B←B,(' '≠¯1↑B)/' ' ⍝ ENSURE A SPACE BETWEEN PROMPT
       AND RESPONSE
[5]    ⍞←B ⋄ A0 ⋄ R←⌊(C+⍴Z←,⍞)÷10 ⋄ ⎕ARBOUT R⍴47×2≥⍳28
       ⍝ ALLOW IDLES FOR RETURN
[6]    →NEXTL IF 0≠⍴Z ⋄ A←((C≠⍴A)×1+C)↓A ⋄ →L1 ⍝ POP A
       LEVEL OF PROMPT
[7]    →NEXTL IF '⎕'≠1↑Z ⋄ Z←⍎⎕ ⋄ →0 ⍝ ALLOW MASS INPUT
[8]    →0 IF '⍞'≠1↑Z ⋄ ⍞←'INTENTIONAL ' ⋄ A0 ⍝ DEBUGGING AID
[9]    RESUME VIA  →GOON
    ∇
```

Note that the prompt is retrieved with the subroutine *PREAD*, which can be as simple or elaborate as desired (file tie checking, validating the prompt number, and so on).

Another spin-off of the multilevel prompting scheme is for the benefit of new or inexperienced users -- a "minimum prompt level". Since they probably won't understand the briefest level of prompts, there is no reason to make these users sit through those prompts and press RETURN each time to elicit a more verbose prompt. Rather, at the beginning of the session, one sets a global variable, say *MPL*, and uses it as in the function *PR6B*:

```
    ∇ Z←PR6B P;A;B;C;R
[1]  GOON:C←0,(P='⍝')/⍳⍴P ⋄ A←C[MPL⌊⍴C]↓P ⍝ ENFORCE A
       MINIMUM PROMPTING LEVEL
[2]  L1:C←⍴B←((A⍳'⍝')-⍳1)↑A ⍝ SPLIT OUT FIRST PROMPT
       IN STRING
[3]  B←B,(' '≠¯1↑B)/' ' ⍝ ENSURE A SPACE BETWEEN PROMPT
       AND RESPONSE
[4]  ⎕←B ⋄ A0 ⋄ R←⌊(C+⍴Z←,⎕)÷10 ⋄ ⎕ARBOUT R⍴47×2≥⌶28
       ⍝ ALLOW SUFFICIENT IDLES FOR RETURN
[5]  →NEXTL IF 0≠⍴Z ⋄ A←((C≠⍴A)×1+C)↓A ⋄ →L1 ⍝ POP A
       LEVEL OF PROMPT (UNLESS IT'S THE LAST ONE)
[6]  →NEXTL IF '⎕'≠1↑Z ⋄ Z←⍎⎕ ⋄ →0 ⍝ ALLOW MASS INPUT
[7]  →0 IF '⍞'≠1↑Z ⋄ ⎕←'INTENTIONAL ' ⋄ A0 ⍝ DEBUGGING AID
[8]  RESUME VIA  →GOON
    ∇
```

Using the example being developed, the minimum prompting level might be used in the following manner:

```
        MPL←2
        PR6B PMT1
NAME AND ADDR ⎕
⎕:
        RESP1
```

If the application spans several workspaces, *MPL* and any other globals needed in the next workspace can be stored on a file unique to each user and retrieved to the workspace as part of the autostart process.

Sooner or later, some users of an application become so proficient with the package that even sitting through the briefest prompts becomes boring. These people would like the ability to enter responses to prompts they have not yet received -- "stack" the input, so to speak. Then, they would need only occasional prompts as signposts so as not to lose track of where they are in the package. This feature is implemented with the addition of only two lines of code and a change in a third. Note, however, that another global variable (*IP*) is required.

```
    ∇ Z←PR7 P;A;B;C;R
[1]  L0:→GOON IF 0=⍴IP ◊ Z←((C←IP⍷'/')-⍳1)↑IP ◊ IP←C↓IP ◊
       →0 ⍝ PROCESS FIRST RESPONSE FROM STACKED INPUT
[2]  GOON:A←,P
[3]  L1:C←⍴B←((A⍷'⍝')-⍳1)↑A ⍝ SPLIT OUT LOWEST LEVEL PROMPT
[4]    B←B,(' '≠¯1↑B)/' ' ⍝ ENSURE A SPACE BETWEEN PROMPT
       AND RESPONSE
[5]    ⍞←B ◊ A0 ◊ R←⌊(C+⍴Z←,⍞)÷10 ◊ ⎕ARBOUT R⍴47×2≥⍳28
       ⍝ ALLOW SUFFICIENT IDLES FOR RETURN
[6]    →NEXTL IF 0≠⍴Z ◊ A←((C≠⍴A)×1+C)↓A ◊ →L1 ⍝ POP
       A LEVEL OF PROMPT (UNLESS IT'S THE LAST)
[7]    →NEXTL IF '⎕'≠1↑Z ◊ Z←⍎⎕ ◊ →L2 ⍝ ALLOW MASS INPUT
[8]    →L2 IF '⍞'≠1↑Z ◊ ⍞←'INTENTIONAL ' ◊ A0 ⍝ DEBUGGING AID
[9]    RESUME VIA  →GOON
[10] L2:IP←IP,Z,('/'≠¯1↑Z)/'/' ◊ →L0 ⍝ APPEND LATEST
       RESPONSE TO STACKED INPUT.  GO PROCESS IT.
    ∇
```

An example of the usefulness of this last enhancement is a data entry application where the user is normally prompted for several pieces of information in each record. The next two samples show terminal sessions without "input stacking" and with it.

```
ACCT NO: 10257-3
NAME LAWRENCE COHEN
ADDR 1267 ROCK ROAD, GENEVA, NY
INIT. AMT. 1123.50
ACCT NO: [and so on]
```

```
ACCT. NO: 10257-3/LAWRENCE COHEN/1267 ROCK ROAD, GENEVA, NY
INIT. AMT. 1123.50/11350-2/IRENE DURHAM
ADDR [and so on]
```

Now the function can be cleaned up and recoded slightly into its "final form" (were that such a thing existed!), which is displayed on the next page.

```
∇ Z←AKI P;A;B;C;R
[1]  L0:→NEXTL IF 0=⍴IP ◊ Z←(C←(IP⍳'/')-⍳1)↑IP ◊ IP←(C+1)↓IP
      ◊ →0 ⍝ STACKED INPUT
[2]   →GOON IF ' '=1↑0⍴P ◊ P←PREAD P ⍝ FILED PROMPT.
      RETRIEVE IT
[3]  GOON:C←0,(P∊'⍝')/⍳⍴P ◊ A←C[MPL⌊⍴C]↓P ⍝ ENFORCE A
      MINIMUM PROMPTING LEVEL
[4]  L1:C←⍴B←((A⍳'⍝')-⍳1)↑A ⍝ SPLIT OUT FIRST PROMPT
[5]   B←B,(' '≠¯1↑B)/' ' ⍝ ENSURE A SPACE BETWEEN PROMPT
      AND RESPONSE
[6]   ⎕←B ◊ A0 ◊ R←⌊(C+⍴Z←,⍞)÷10 ◊ ⎕ARBOUT R⍴47×2≥⍳28
      ⍝ ALLOW IDLES FOR CARRIER RETURN
[7]   →NEXTL IF 0≠⍴Z ◊ A←((C≠⍴A)×1+C)↓A ◊ →L1 ⍝ POP A
      LEVEL OF PROMPT (EXCEPT LAST)
[8]   →(L2,L3,L4)['⎕⍞'⍳1↑Z]
[9]  L2:Z←⍎⎕ ◊ →0 ⍝ ALLOW MASS INPUT
[10] L3:⎕←'INTENTIONAL ' ◊ A0 ⍝ DEBUGGING AID
[11]  RESUME VIA  →GOON
[12] L4:IP←IP,Z,('/'≠¯1↑Z)/'/' ◊ →L0 ⍝ APPEND LATEST
      RESPONSE(S) TO INPUT STACK.  GO PROCESS.
∇
```

Note that this "final form" of the prompting function is now called *AKI* for Accept Keyboard Input.

Prompting for Numeric Input

What features are desired in a function that prompts for numeric input as opposed to one that prompts for character input? First, a function that prompts for numeric input should perform all the checks that the *AKI* function, developed in the previous section, performs. Additionally, it should check that what is received really translates into numbers -- that is, it passes the "⎕*VI* test".

```
    ∇ Z←PN1 P
[1]  L1:Z←AKI P ◊ →NEXTL IF∧/⎕VI Z ◊ '*** NON-NUMERIC
       CHARACTERS IN RESPONSE. TRY AGAIN' ◊ →L1
[2]    Z←⎕FI Z
    ∇
```

Now, as with the development of the character prompting function, some primitive error-checking capabilities can be built into the numeric prompting function. For instance, many functions that prompt for numeric input are looking for a specific number of responses.

```
    ∇ Z←N PN2 P;A
[1]  L1:Z←AKI P ◊ →NEXTL IF∧/A←⎕VI Z ◊ '*** NON-NUMERIC
       CHARACTERS IN RESPONSE. TRY AGAIN' ◊ →L1
[2]    →NEXTL IF N∊0,+/A ◊ '*** INCORRECT NUMBER OF
       RESPONSES.  WANTS ',⍕N ◊ →L1
[3]    Z←⎕FI Z
    ∇
```

Note that if the left argument is zero, the function accepts any number of valid (i.e., numeric) responses. If *N* is nonzero, the function accepts exactly that many responses. All others result in error messages.

The next enhancement one might add is to allow a flexibility in the APL syntax. Inexperienced users may be unfamiliar with the difference between "middle minus" and the APL "high minus". Rather than having an instructor try to teach the concept, or even teach the method by rote, the function allows either and equates both to the APL "high minus". Similarly, there is a tendency among people trained in certain disciplines to separate numeric entries with commas. These can be made to map into spaces for conformability with the rules of ⎕*FI* and ⎕*VI*. The scheme is to build a translation table like this:

```
B←'  0123456789¯¯E.*'[' ,0123456789-¯E.'⍳Z]
```

where *Z* is the output of *AKI*. An alternative or modification to this table would also map lower-case "L" into "1" for people who are not used to typewriters with a "1" key:

```
B←'  01123456789¯¯E.*'[' ,01L23456789-¯E.'⍳Z]
```

Note that, in both cases, all illegal characters are mapped into "*". (The "E" is present in both translate tables to allow the use of scientific notation.) Thus, their positions are easily determined and pointed out in diagnostic messages. The function *PN3* incorporates just such a scheme. Following the example introduced in the previous section, an incorrect entry of "*INIT. AMT.*" is indicated as follows:

```
      1 PN3 'INIT. AMT.'
INIT. AMT. 1120:25
*** INVALID CHARACTER
1120:25
    ^
INIT. AMT. [and so on]
```

```
    ∇ Z←N PN3 P;A;B
[1]  L1:Z←AKI P ◊ B←'  0123456789¯¯E.*'
       [' ,0123456789-¯E.'⍳Z]
[2]   →L2 IF∧/A←⎕VI B ◊ B←(B∊'*')\'∧' ◊ '*** INVALID
      CHARACTER','S' IF 1<+/B≠' '
[3]   Z ◊ B ◊ →L1 ⍝ DISPLAY INPUT ◊ MARK INVALID CHARS
[4]  L2:→L3 IF N∊0,+/A ◊ '*** INCORRECT NUMBER OF RESPONSES.
      WANTS ',⍕N ◊ →L1
[5]  L3:Z←⎕FI B
    ∇
```

This, however, solves only part of the problem. What happens with a character sequence like "12¯3"? ⎕*VI* recognizes it as garbage but, since the translated representation contains no "*"s, the diagnostic message is incomplete. What is desired is to be able to find either type of error. With a little bit of software magic, this can be done.

```
    ∇ Z←N PN3A P;A;B;C
[1]  L1:Z←AKI P ◊ B←'  0123456789¯¯E.*'
       [' ,0123456789-¯E.'⍳Z]
[2]   →L2 IF∧/A←⎕VI B ◊ C←Z∊' ,' ◊ C←(C<1,¯1↓C)\~A
[3]   '*** INVALID ',(6-15×'*'∊B)↑'NUMBERCHARACTER' ◊ Z
      ◊ C\'∧' ◊ →L1 ⍝ DISPLAY INPUT ◊ MARK INVALID CHARS
[4]  L2:→L3 IF N∊0,+/A ◊ '*** INCORRECT NUMBER OF RESPONSES.
      WANTS ',⍕N ◊ →L1
[5]  L3:Z←⎕FI B
    ∇
```

Now, all the errors discovered by ⎕*VI* are displayed and properly marked in the diagnostic message.

Another feature which is both useful and easy to implement (those are the best kind) allows a varying number of responses. Where *PN2* allows exactly *N* responses (*N*>0), it is a simple expansion to allow *N* to be a two-element vector indicating limits to the number of responses. However, a single number still works as a special case of this extension.

```
    ∇ Z←N PN4 P;A;B
[1]   N←N+ 0 1 ⍝ TO ALLOW FOR A RANGE OF RESPONSES
[2]  L1:Z←AKI P ⋄ B←'  0123456789¯¯E.*'
       [' ,0123456789-¯E.'⍳Z]
[3]   →L2 IF∧/A←⎕VI B ⋄ B←(B∊'*')\'∧' ⋄ '*** INVALID
       CHARACTER','S' IF 1<+/B≠' '
[4]   Z ⋄ B ⋄ →L1 ⍝ DISPLAY INPUT ⋄ MARK INVALID CHARS
[5]  L2:→L3 IF((+/A)≠.<N)∨0∊N ⋄ B←'*** INCORRECT NUMBER OF
       RESPONSES. WANTS ',⍕1↑N
[6]   B,((¯1>-/N)/' TO ',(⍕¯1+¯1↑N)),' NUMBERS' ⋄ →L1
[7]  L3:Z←⎕FI B
    ∇
```

If *N* is a two-element vector, the left argument of the compression in line [6] of *PN4* is a 1, and the full diagnostic is displayed. If *N* is a single number (when passed to *PN4*), an abbreviated error message similar to that displayed by *PN3* is produced. Since it might be confusing, the diagnostic message can (and probably should) be modified to actually list the number of responses that may be entered.

```
    ∇ Z←N PN4A P;A;B;C
[1]   N←0 1+(>/N)⌽N
[2]  L1:Z←AKI P ⋄ B←'  0123456789¯¯E.*'
       [' ,0123456789-¯E.'⍳Z]
[3]   →L2 IF∧/A←⎕VI B ⋄ B←(B∊'*')\'∧' ⋄ '*** INVALID
       CHARACTER','S' IF 1<+/B≠' '
[4]   Z ⋄ B ⋄ →L1 ⍝ DISPLAY INPUT ⋄ MARK INVALID CHARS
[5]  L2:→L3 IF((+/A)≠.<N)∨0∊N ⋄ '*** INCORRECT NUMBER OF
       RESPONSES' ⋄ A←N[⍳1]+¯1+⍳-/⌽N
[6]   B←'ENTER ',(⍕¯1↓A),(1≠⍴A)/' OR ' ⋄ B,(⍕¯1↑A),
       ' NUMBERS' ⋄ →L1
[7]  L3:Z←⎕FI B
    ∇
```

Now, the "final form" of the function is put together, the logic cleaned up, and the function renamed *NAKI* (*ANKI* might be a more consistent name but historical accidents are a strong precedent). *NAKI* is displayed on the following page.

```
    ∇ Z←N NAKI P;A;B;C
[1]   N←N+ 0 1 ⍝ TO ALLOW FOR A RANGE OF RESPONSES.
      SIMILARLY  N←N+(≤/2⍴N)⌽1 0
[2]  L1:Z←AKI P ⋄ B←'  0123456789¯¯E.*'
      [' ,0123456789-¯E.'⍳Z] ⍝ TRANSLATE TO CANONICAL FORM
[3]   →L2 IF∧/A←⎕VI B ⋄ C←Z∊' ,' ⋄ C←(C<1,¯1↓C)\~A
      ⍝ LOCATE START OF EACH ILLEGAL FIELD
[4]   '*** INVALID ',(6-15×'*'∊B)↑'NUMBERCHARACTER' ⋄ Z
      ⋄ C\'^' ⋄ →L1 ⍝ DIAGNOSTIC MSG FOR ILLEGAL RESPONSE
[5]  L2:→L3 IF((+/A)≠.<N)∨0∊N ⋄ '*** INCORRECT NUMBER OF
      RESPONSES' ⋄ A←N[⍳1]-(⍳1)-⍳-/⌽N ⍝ SIMILARLY
      A←(⌊/N)-(⍳1)-⍳|-/N
[6]   B←'ENTER ',(⍕¯1↓A),(1≠⍴A)/' OR ' ⋄ B,(⍕¯1↑A),
      ' NUMBER','S' IF 1∨.≠A ⋄ →L1 ⍝ WRONG NUMBER OF
      RESPONSES
[7]  L3:Z←⎕FI B ⍝ PASSED THE TESTS
    ∇
```

Prompting for a Specific Response

Another extension to the suite of prompting routines is a function that compares the user's response to a list of acceptable responses and returns the index of the response in the list. If the response is not found in the list, a number of actions are possible, as seen in the discussion that follows.

In function *AN1* below, the matrix of acceptable responses is contained in the global variable *AM*. The result of a call to *AKI* is compared to the answer matrix. If there is a match, the index of the response (its row number in *AM*) is returned. If there is no match, the response itself is passed as the result.

```
    ∇ Z←AN1 P;A
[1]   Z←AKI P ◊ Z←(1↓⍴AM)↑Z ◊ A←AM∧.=Z
[2]   →0 IF~∨/A ◊ Z←A/⍳⍴A
    ∇
```

The first simple enhancement that can be made to this function is to pull the matrix of acceptable responses "inboard" and make it the left argument. This doesn't take any more space, and it makes the function a little more flexible because it is more self-contained.

```
    ∇ Z←AM AN1A P;A
[1]   Z←AKI P ◊ Z←(1↓⍴AM)↑Z ◊ A←AM∧.=Z
[2]   →0 IF~∨/A ◊ Z←A/⍳⍴A
    ∇
```

In most applications where one would want to limit the responses to a known set, it is usually the case that responses outside of that set are rejected with some sort of diagnostic message, and the prompt repeated until an acceptable response is encountered. As it is currently written, *AN1A* relies on the calling function to do this checking. It makes a great deal of sense to have the prompting function itself perform this check and it is fairly simple to make this modification. In fact, the function can easily be made to list the valid responses.

```
    ∇ Z←AM AN2 P
[1]  L1:Z←AKI P ◊ Z←AM∧.=(1↓⍴AM)↑Z
[2]   →NEXTL IF~∨/Z ◊ Z←Z/⍳⍴Z ◊ →0
[3]   '*** INVALID RESPONSE' ◊ 'VALID ANSWERS ARE  ',
      ' ' SQ ,AM,' ' ◊ →L1
    ∇
```

The call to *SQ* in the diagnostic message keeps the message as short as possible by eliminating excess spaces on the trailing end of the shorter valid responses.

Even this function is limited in one major respect -- it handles only character-valued data. The extension to allow it to handle numeric data is straightforward, although nontrivial.

First, the function must be able to determine the type of data it's supposed to be processing. The expression 0=1↑0⍴AM provides this determination. A 1 as the result indicates type "numeric"; a 0 indicates type "character". If character data is indicated, processing continues as it did in the previous examples. If the matrix of valid answers is numeric, the standard tests are performed on the response and processing continues after the response is converted to type "numeric".

```
      ∇ Z←AM AN2A P
[1]     AM←(2⍴(⍴AM),1)⍴AM ⍝ ENSURE A MATRIX
[2]    L1:Z←AKI P ⋄ →NEXTL IF 0≠1↑0⍴AM ⋄ →L3 IF~∧/⎕VI Z
         ⋄ Z←⎕FI Z ⋄ →L3 IF(⍴Z)≠¯1↑⍴AM
[3]     Z←AM∧.=(¯1↑⍴AM)↑Z ⋄ →L2 IF∨/Z
[4]    L3:'*** INVALID RESPONSE' ⋄ 'VALID ANSWERS ARE  ',
         ¯1↓' ' SQ,' ',(⍕AM),',' ⋄ →L1
[5]    L2:Z←Z/⍳⍴Z
      ∇
```

Note that in this version of the routine, when the list of acceptable responses is displayed, commas are placed between them because a single space cannot be recognized as a separator in the case of numeric-vector choices. (This is also a nice touch in the case of character-valued responses when there may be embedded blanks in a valid response.)

Another alternative in the development of this type of prompting function is the use of *NAKI* when numeric input is desired. This function provides all the error checking done in the previous example and improved error diagnostics, to boot.

```
      ∇ Ż←AM AN3 P;A
[1]     AM←(2⍴(⍴AM), 1 1)⍴AM ⋄ →(A←0=1↑0⍴AM)↓L1,L2
         ⍝ ENSURE A MATRIX. DETERMINE DATA TYPE.
[2]    L1:Z←AKI P ⋄ →L3
[3]    L2:Z←0 NAKI P
[4]    L3:Z←AM∧.=(¯1↑⍴AM)↑Z ⋄ →0 IF 0≠⍴Z←Z/⍳⍴Z ⍝ COMPARE
         TO VALID RESPONSES. QUIT IF A MATCH.
[5]     '*** INVALID RESPONSE' ⋄ 'VALID ANSWERS ARE  ',
         (-A)↓' ' SQ,(⍕AM),''⍴A↓' ,' ⋄ →A↓L1,L2
      ∇
```

Finally, another small bit of implicit error checking can be added to the function's handling of numeric input.

```
      ∇ Z←AM ANS P;A
[1]     AM←(2⍴(⍴AM), 1 1)⍴AM ⋄ →(A←0=1↑0⍴AM)↓L1,L2
         ⍝ ENSURE A MATRIX. DETERMINE DATA TYPE.
[2]    L1:Z←AKI P ⋄ →L3
[3]    L2:Z←(1,¯1↑⍴AM) NAKI P ⍝ ALLOW UP TO ¯1↑⍴AM ELEMENTS
[4]    L3:Z←AM∧.=(¯1↑⍴AM)↑Z ⋄ →0 IF 0≠⍴Z←Z/⍳⍴Z ⍝ COMPARE
         TO VALID RESPONSES. QUIT IF A MATCH.
[5]     '*** INVALID RESPONSE' ⋄ 'VALID ANSWERS ARE  ',
         (-A)↓' ' SQ,(⍕AM),''⍴A↓' ,' ⋄ →A↓L1,L2
      ∇
```

Data Security

The problem most often encountered in writing interactive programs is an inexperienced user's accidentally clobbering some vital information stored in the workspace, thereby either rendering the package in the active workspace inoperable or (more seriously) causing erroneous information to be entered into the database. The next most frequent problem, and one far more difficult to deal with, is a user's intentionally changing variables in the workspace in an attempt to modify data in a manner other than that intended by the designer.

The simplest security technique is to perform all input from the terminal through quote-quad input. This effectively prevents the inexperienced user from getting into too much trouble since he is separated from the outside world by an implicit set of quotes. This also provides a first level of protection from the casual tinkerer who might try to damage the application package by looking at *)SIV* and changing a few local variables while in evaluated input mode.

More substantial measures are needed to protect against a determined assault on security. The most obvious techniques are passnumber protection of files and selective or general locking of the functions in the workspace. (It should be noted here that a file passnumber should <u>never</u> be assigned to a variable. It should <u>always</u> be hardwired into all the functions that actually perform file I/O.) The next logical step, beyond locking functions is using the *)SEAL* command on the workspace. This prevents a determined code-breaker from replacing one of your functions with one of his own and, thereby, peeking at and possibly changing some local variables (which is known in the idiom as "Trojan Horse"-ing).

Still another means of securing an application is by setting *CLEAROUT* at critical points in the code. If the entire workspace is a single autostarted package, it may be desirable to set *CLEAROUT* as the first statement of the autostart line and never reset it. Setting *CLEAROUT* may not be a feasible course of action, because of possible error conditions which, if *CLEAROUT* were set, would be impossible to track down. Several alternatives are available.

It is possible to condition a function so that it does not execute properly except when called as a subroutine. If the following line of code is placed on the first line of the function, it exits immediately unless there is an active, pendent (not suspended) function immediately "above" it in the execution stack:

```
→0 IF 1=+/~∨\∨/⎕SI∊'*⍎⎕'
```

Another method is to check ⎕*AI*[4], keyboard unlock time (origin 1), at various locations in the functions. Note that

this should not be done in places where input is accepted from the terminal. In other situations, however, it gives an indication that, for whatever reason, execution has been suspended and resumed. If the functions are sufficiently paranoid (or their authors overly security conscious), execution is abandoned or some other drastic action taken. An example of the use of ⎕AI[4] for security checking is given below:

```
I19←⎕AI[4]
  o
  o
critical section of code
  o
  o
→NEXTL IF I19=⎕AI[4] ◊ panic processing
```

Actually, ⎕AI[4] checks could encompass programmed keyboard unlocks if this is taken into account in the programs. For example,

```
... ◊ T←⎕AI[4] ◊ R←⍞ ◊ I19←I19+⎕AI[4]-T ◊ ...
```

adds the duration of the programmed unlock to the initial setting of I19.

If the workspace in which checks of keyboard unlock time are performed is not conditioned with *CLEAROUT*, it is strongly suggested that a symbol less obvious than I19 be used to store the value of keyboard unlock time. It's somewhat self-defeating to build a security measure into a system and then advertise its existence. (The source of keyboard unlock time was previously I19.)

There are, of course, alternatives to panic processing. A straightforward alternative is to re-execute a portion of code whenever a check of keyboard unlock time indicates possible impeachment of data. The question arises, "How do we know where to resume?". It is necessary to build guideposts into the function in order to know how far we must backtrack in this situation. The local variable RST is used for this purpose.

```
RST←1↑⎕LC ⍝ OPTIONALLY,  RST←THISL
  o
  o
R←AKI 'LAST PROMPT IN GROUP'
I19←⎕AI[4]
  o
  o
secured calculations
  o
  o
→NEXTL IF I19=⎕AI[4] ◊ '*** WARNING.  SUSPENSION HAS
      OCCURRED.  UPDATE RESTARTED.' ◊ →RST
I19←⎕AI[4] ◊ CN1 REPLACE DATA1 ◊ CN2 REPLACE DATA2 ◊ ...
      ◊ →RST IF I19≠⎕AI[4]
```

The checks of keyboard unlock time ensure that there is no possibility of undetected tampering with the data during critical computations or synchronous file updates. Knowing the setting of *RST*, we know how far to backtrack in order eliminate the effects of any such tampering.

The *RST*←1↑⎕*LC* is also a helpful signpost in restarting after a line drop. If file tie checks are not necessary, something as simple as →*RST* could be sufficient for restarting.

Restarting Functions

In previous sections, the philosophy of presentation has been that one could take a bare-bones function that performs a stated task, enlarge upon it, add enhancements, improve error diagnostics, and the like. A discussion of restart functions does not lend itself to this approach. There are so many actions that might be performed by such a function that a more reasonable course is to give examples of a few of the more common algorithms currently in use. In all cases, the syntax of the restart function is

```
→RESTART
```

This implies, of course, that the line on which execution is to resume must be returned as an explicit result.

The simplest form of a restart function merely picks up where execution was suspended. It performs no error checking and does not retie files.

```
    ∇ Z←RS1
[1]   Z←1↑1↓⎕LC
    ∇
```

This is an obvious base from which to build more elaborate functions and the single statement it contains is found, in one form or another, in almost every restart function written. A simple embellishment to this function takes advantage of the variable RST, defined in the section "Data Security". If RST is defined in the suspended function, its value is used as the number of the line to be restarted. If it doesn't exist, the current line is used as the restart point.

```
    ∇ Z←RS2
[1]   Z←1↑1↓,⎕IDLOC 'RST' ◊ Z←2≠Z ⍝ IS RST DEFINED IN THE
      'CALLING' FN?
[2]   Z←Z↓RST,1↑1↓⎕LC ⍝ IF SO, USE IT; ELSE, USE CURRENT
      LINE
    ∇
```

Another addition that could be made to the restart function is the retying of files. This may or may not be in the domain of a restart function, depending on how file I/O is done. If a function such as *PREAD*, shown next, is used for file reads, it is not necessary for the restart function to retie files. File ties are checked before each file operation and appropriate tying and untying of files done as necessary. This is probably the least error-prone of all tie-checking schemes and only marginally more expensive (due to more ⎕*FNAMES* and ⎕*FNUMS* operations in the *TI* function).

```
    ∇ Z←PREAD CN
[1]   'PRFILE' TI PTN ◊ Z←⎕FREAD PTN,CN
    ∇
```

If this tie-checking method is not used and the responsibility is placed on the restart function, then the names and tie numbers for the files to be retied must reside in a global variable in the workspace. In the function *RS3*, it is assumed that the variable *NAMES* is a matrix of the form produced by a ⎕*FLIB* or ⎕*FNAMES* operation. Note that *NAMES*[*I*;] is tied to *NUMS*[*I*] in this and most other systems. In this case, a rather novel technique for retying files has been developed.

```
    ∇ Z←RS3;I;N;T
[1]  ⍝ DETERMINE WHOSE FILE IT IS. FTIE OURS, FSTIE OTHERS
[2]   T← 0 ¯11 ↓NAMES ◊ N←(∧/T∊' ')/⍳⍴NUMS
[3]   T[N;⍳10]←'I10' ⎕FMT(⍴N)⍴⍉29 ◊ T←2+2×(1↑⎕AI)≠⎕FI,T
[4]   Z←1↑1↓⎕LC ◊ I←1 ◊ N←⍴NUMS
[5]  L1:→0 IF N<I ◊ NAMES[I;] FF T[I],NUMS[I] ◊ I←I+1 ◊ →L1
    ∇
```

Another minor facility that may be desirable is the handling of restart when the local environment is evaluated input. It is not impossible for a line drop to occur during or just prior to the appearance of a ⎕: prompt. An inexperienced user may not notice (or may not understand) the presence of the prompt when he signs back on. A →*RESTART* at this point causes a *VALUE ERROR* report, further adding to the confusion. In this situation a special check must be made for 0=1↓⎕*LC* and special action taken if the condition is true.

```
    ∇ Z←RS4
[1]   ⎕←'RESTART INITIATED' ◊ →L1 IF 1=⍴⎕LC ◊
      →L1 IF 0≠1↑1↓⎕LC
[2]   ⎕←'  (IGNORE THE ERROR)' ◊ →0
[3]  L1:Z←1↑1↓⎕LC
    ∇
```

The error condition that the function says to ignore is the *VALUE ERROR* report caused by the absence of an explicit result being returned from the branch arrow function. In this manner, the function does everything it has to (in this case, nothing), and the integrity of the ⎕: is retained.

Validating Input

As is the case with "restart" functions, the number of ways to check for valid input is so extensive that no attempt will be made to develop a "final form" of an input validation function. Only two examples which share a common subroutine are examined. The two forms are similar. The first form checks the data for membership in a particular set of values. If all of the data is in that set, the data values are passed as the result of the function. If any values lie outside the specific set, a distinctive value (contained in the variable *INVALID*) is returned from the function and an extensive diagnostic message is issued.

```
    ∇ Z←VALID CHKVALS X;A
[1]   Z←INVALID ⍝ RESET IF AND ONLY IF ALL CHECKS OK
[2]   →L1 IF∧/,A←X∊VALID ◊ '*** UNACCEPTABLE VALUE','S' IF
      1<+/~,A
[3]   X DATAERROR A ◊ →0
[4]  L1:Z←X
    ∇
```

An alternative method of signalling valid versus invalid results is to return a 1 or 0, respectively.

The other common form of input validation checks whether or not all of the data falls within a given range. The exact values of the data are unimportant (to the prompting function, at least) as long as they fall in that range. One application of this function is for "reasonableness" checks on the data.

```
    ∇ Z←VALID RANGE X;A
[1]   Z←INVALID ⍝ RESET IF ALL DATA OK
[2]   A←,(X≥1↑VALID)∧X≤¯1↑VALID ◊ →L1 IF∧/A
[3]   ⎕←'*** NUMBER','S' IF 1<+/~A ◊ ⎕←' OUT OF RANGE'
[4]   X DATAERROR(⍴X)⍴A ◊ →0
[5]  L1:Z←X
    ∇
```

Both of these functions call a subroutine to display the major error diagnostic. The function *DATAERROR* takes the data as its left argument and a bit array of the same shape as its right argument. Zeros in the right argument indicate data that is, for some reason, unacceptable. The diagnostic message consists of the line of data containing the invalid elements and a line with carets pointing to those elements.

```
    ∇ Z←X DATAERROR Y;A;B;C
[1]  ⍝ <X> IS THE RAW DATA (ANY SHAPE)
[2]  ⍝ <Y> IS VALIDITY FLAGS; 1←→OK, 0←→BAD; (⍴Y)←→⍴X
[3]   A←~∧/Y ⋄ X←⍉X ⋄ X←((×/¯1↓⍴X),¯1↑1,⍴X)⍴X ⍝ FORCE A
      MATRIX
[4]   C←X∊' ' ⋄ B←⍴C←C<1, 0 ¯1 ↓C ⋄ C←B⍴(,C)\(~,Y)\'∧'
[5]   Z←A≠X ⋄ B←(2×+/A)⍴ 1 0 ⋄ Z←B⌹Z ⋄ Z[(~B)/⍳⍴B;]←A≠C
    ∇
```

An example of the use of *RANGE* and the subroutine *DATAERROR* follows:

```
      0 1 RANGE 5 NAKI 'PROMPT'
PROMPT .54 .37 4.5 .88 .91
*** NUMBER OUT OF RANGE
0.54 0.37 4.5 0.88 0.91
        ^
```